CS420: Operating Systems

Threads

James Moscola
Department of Physical Sciences
York College of Pennsylvania

Threads

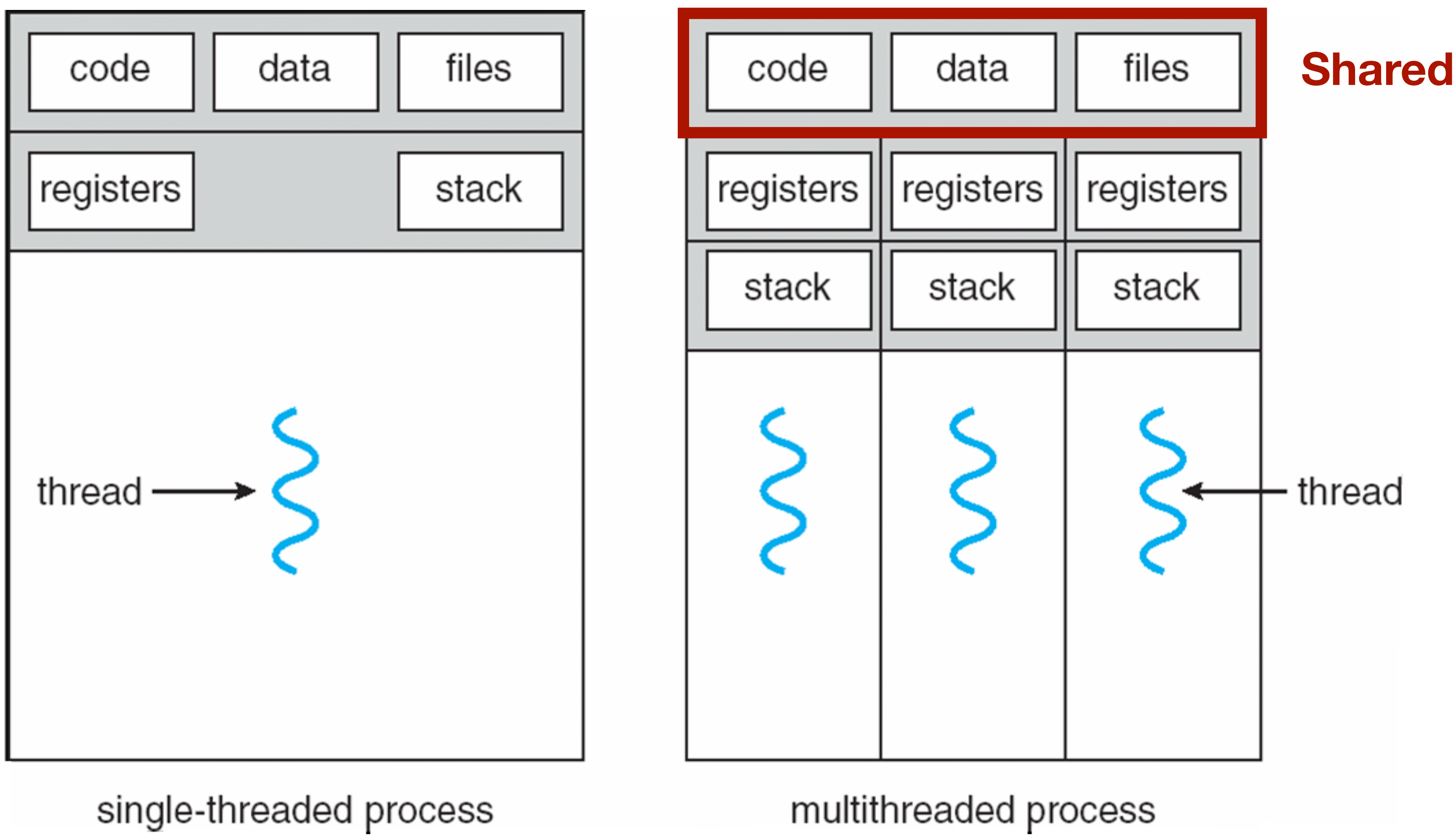
- **A thread is a basic unit of processing**
  - Has the following components:
    - Thread ID
    - Program counter
    - Register set
    - Stack
  - Shares some resources with other threads in same process
    - Code section
    - Data section
    - OS Resources (e.g. open files, signals)
- **Scheduled by the operating system**

Threads

- **A heavyweight process is a process that has a single thread of control**
  - Can only perform a single task at a time

- **A multi-threaded process is a process that has multiple threads of control**
  - Can perform more than one task at a time
    - Render images
    - Fetch data
    - Update display
    - Check spelling

Single and Multithreaded Processes



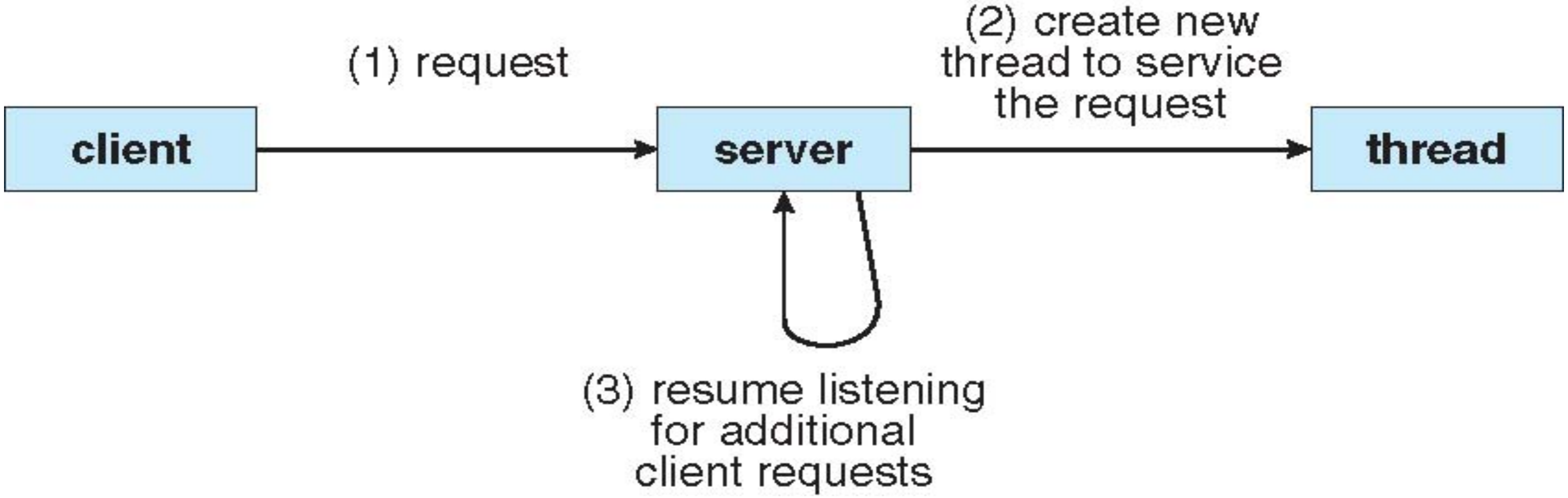
Thread vs Process

- **Processes -**
  - Independent units of execution
  - Each contains its own state information
  - Each contains its own address space
  - Interact with each other through various IPC mechanisms
- **Threads (within the same process) -**
  - Share the same state
  - Share the same memory space
  - Share the same variables
  - Can communicate directly through shared variables
  - Share signal handling

Benefits of Multithreaded Programming

- **Responsiveness**
  - Interactive applications are more responsive when multithreaded (e.g. a thread for the GUI, another for socket, a third for rendering, etc.)
- **Resource Sharing**
  - Unlike processes, threads share memory and resources
- **Economy**
  - Since threads share resources, creating threads and switching between them is more efficient than processes
- **Scalability**
  - Multithreading allows for increased parallelism on multicore systems as each thread can run on a different CPU core

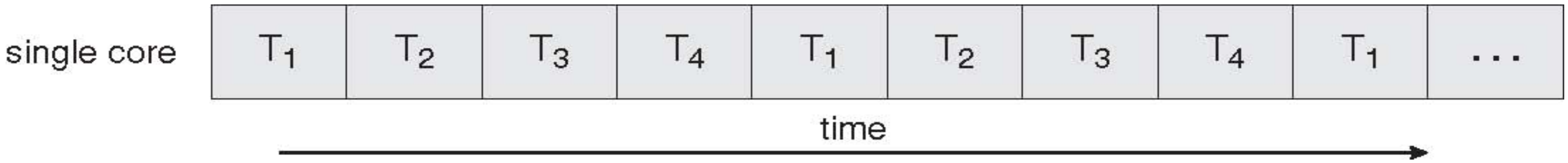
Multithreaded Server Architecture



Multicore Programming

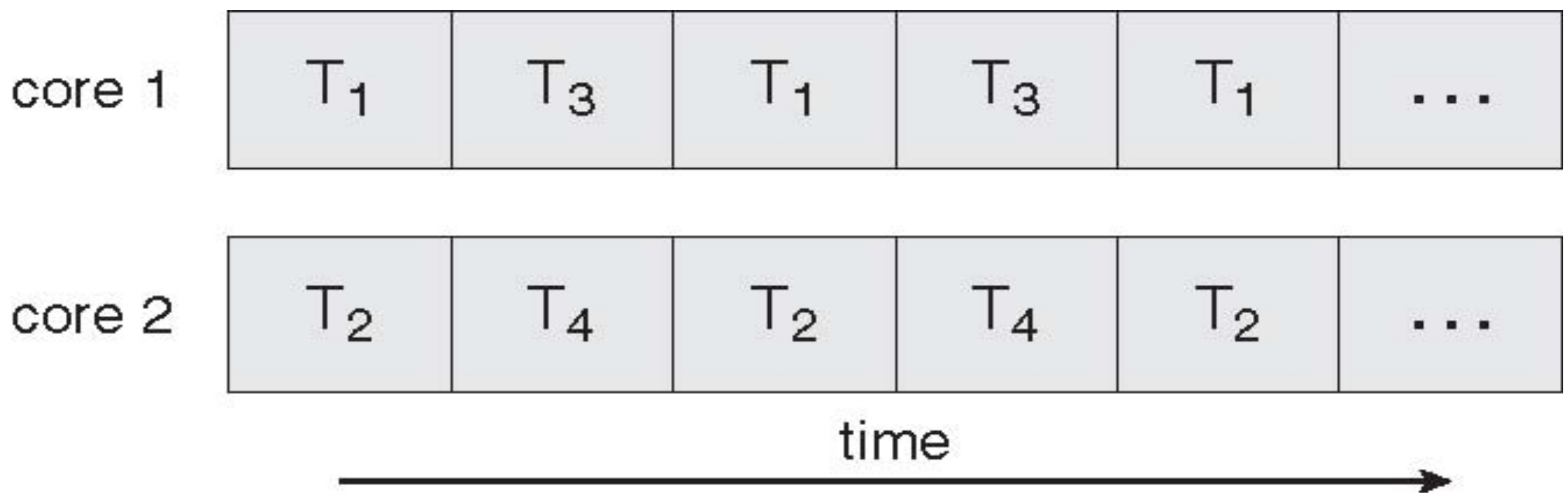
- **The performance of a process can be increased by properly threading the process to take advantage of modern multicore CPUs**
- **Multicore systems have challenges not faced in a single-core/single-threaded environment**
  - **Dividing activities** - How can an application be divided into separate, concurrent tasks?
  - **Balance** - How can those tasks be divided in such a way that each does an equal amount of work?
  - **Data splitting** - Can the data for those task be divided for processing on separate CPU cores?
  - **Data dependency** - Are there data dependencies between different task?
  - **Testing and debugging** - What is the best way to debug a multithreaded program with many different execution paths?

Concurrent Execution on a Single-core System



- **Only a single thread can execute at a time**
- **Threads are interleaved so each gets time on the processor**

Parallel Execution on a Multicore System



- **With multiple cores, threads can be divided over the cores and run in parallel**
- **May still interleave threads if not enough cores are available for all of the threads**

Thread Support

- **Threads may be supported at different levels of the OS**
  - **User threads**
    - Supported above the kernel
    - Managed without kernel support
    - Three main user thread libraries currently in use -
      (1) POSIX PThreads (2) Win32 threads (3) Java threads
  - **Kernel threads**
    - Supported by the kernel/operating system
    - Managed by the kernel/operating system
    - Most modern operating systems support kernel threads
      (e.g. Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X)

Multithreading Models

- **There must be a relationship between user level threads and kernel threads**

- **Different models of threading exist to define this relationship**
  - Many-to-One
  - One-to-One
  - Many-to-Many

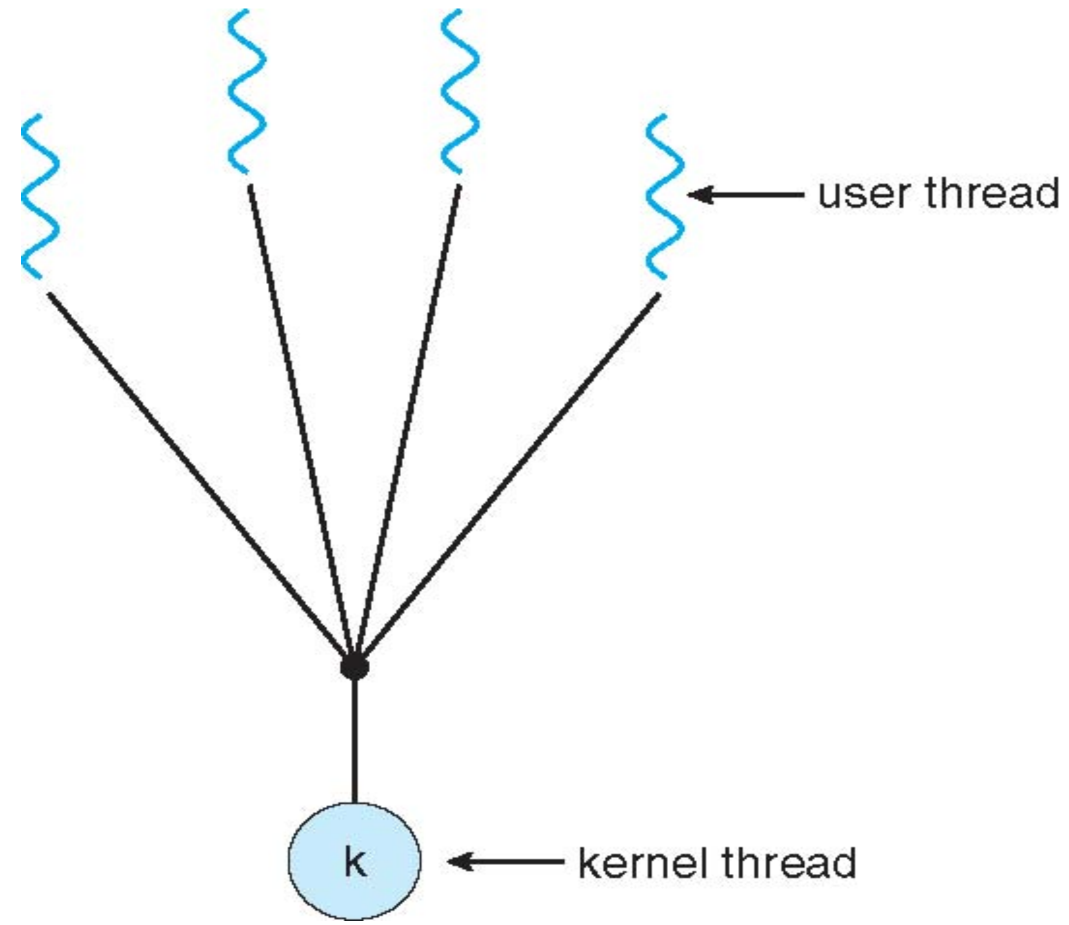
Many-to-One

- **Many user-level threads mapped to single kernel thread**
  - Examples: Solaris Green Threads, GNU Portable Threads

- **Thread management is done in user space**

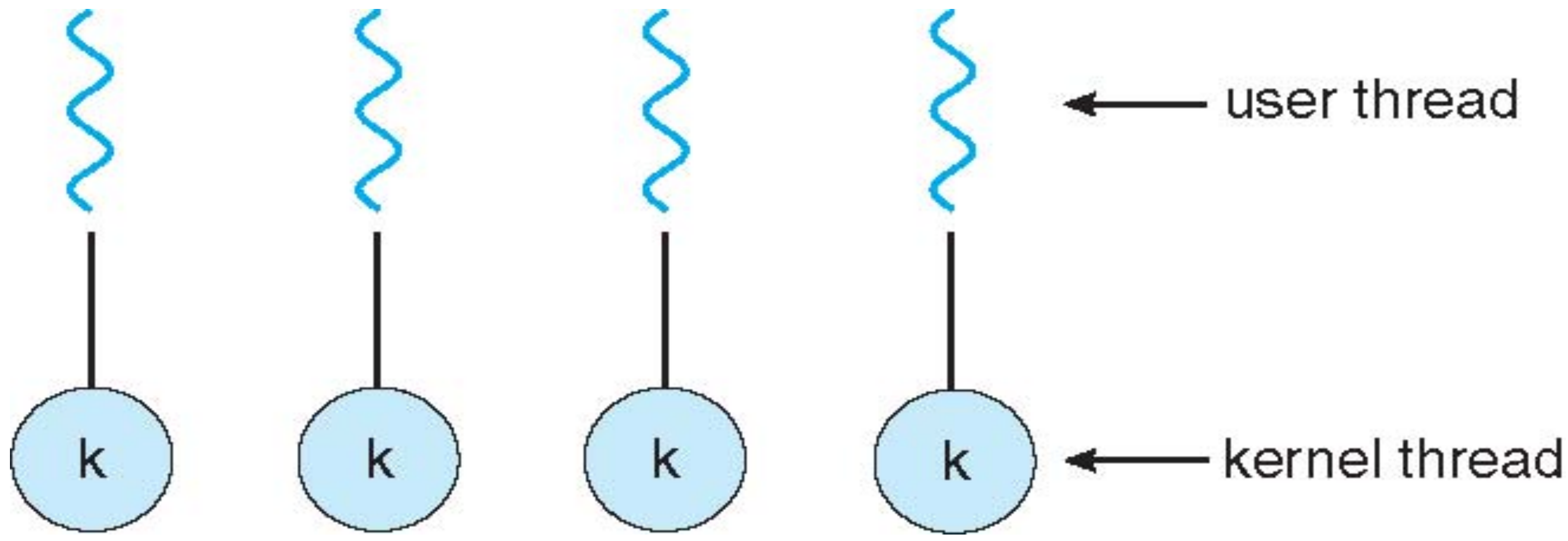
- **Entire process will block if any single thread blocks (no other threads will run)**

- **Unable to run multiple threads in parallel on a multiprocessor system**



One-to-One

- **Each user-level thread maps to kernel thread**
  - Examples: Windows NT/XP/2000, Linux, Solaris 9 and later
- **Allows more concurrency**
  - A thread can run when another thread has made a blocking system call
  - Multiple threads can run in parallel on multiprocessor systems
- **Downside, for each thread created, a corresponding kernel thread must also be created**

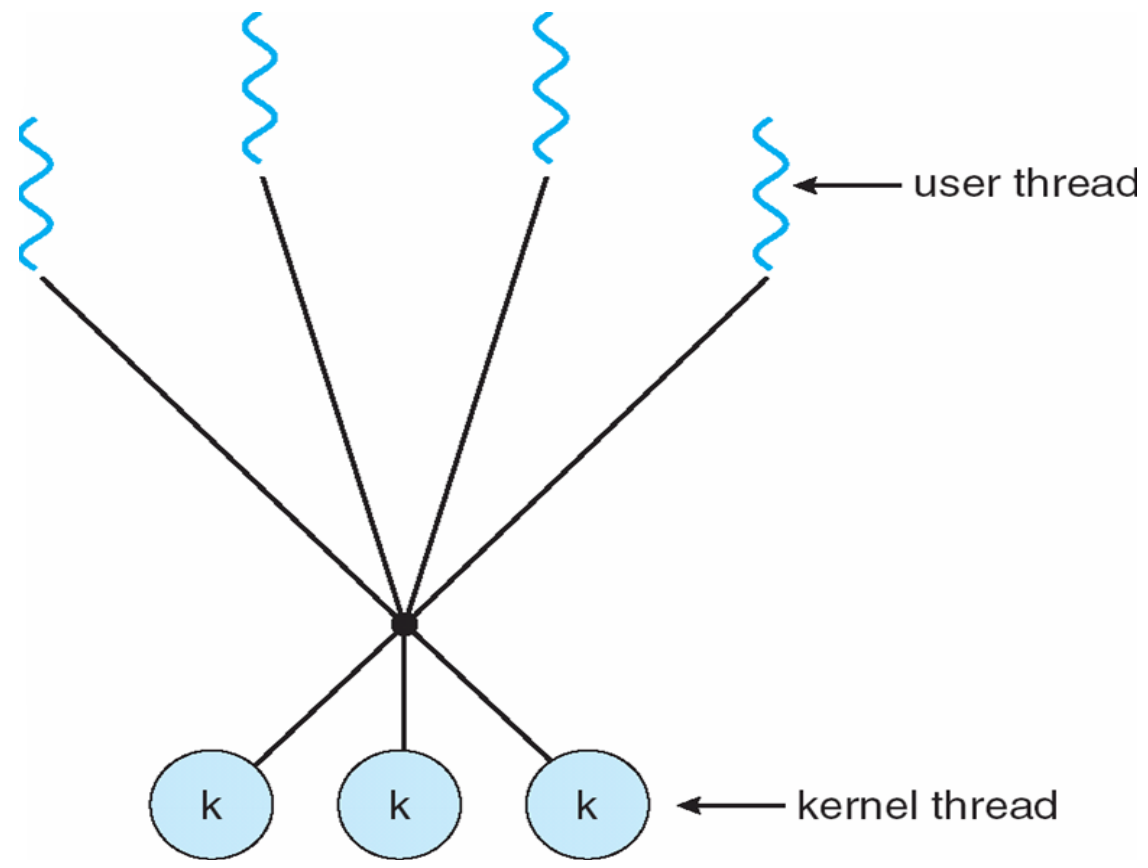


Many-to-Many Model

- **Allows many user level threads to be mapped to many kernel threads**
  - Avoids blocking of threads when a single thread makes a blocking system call

- **Allows the operating system to create a sufficient number of kernel threads**

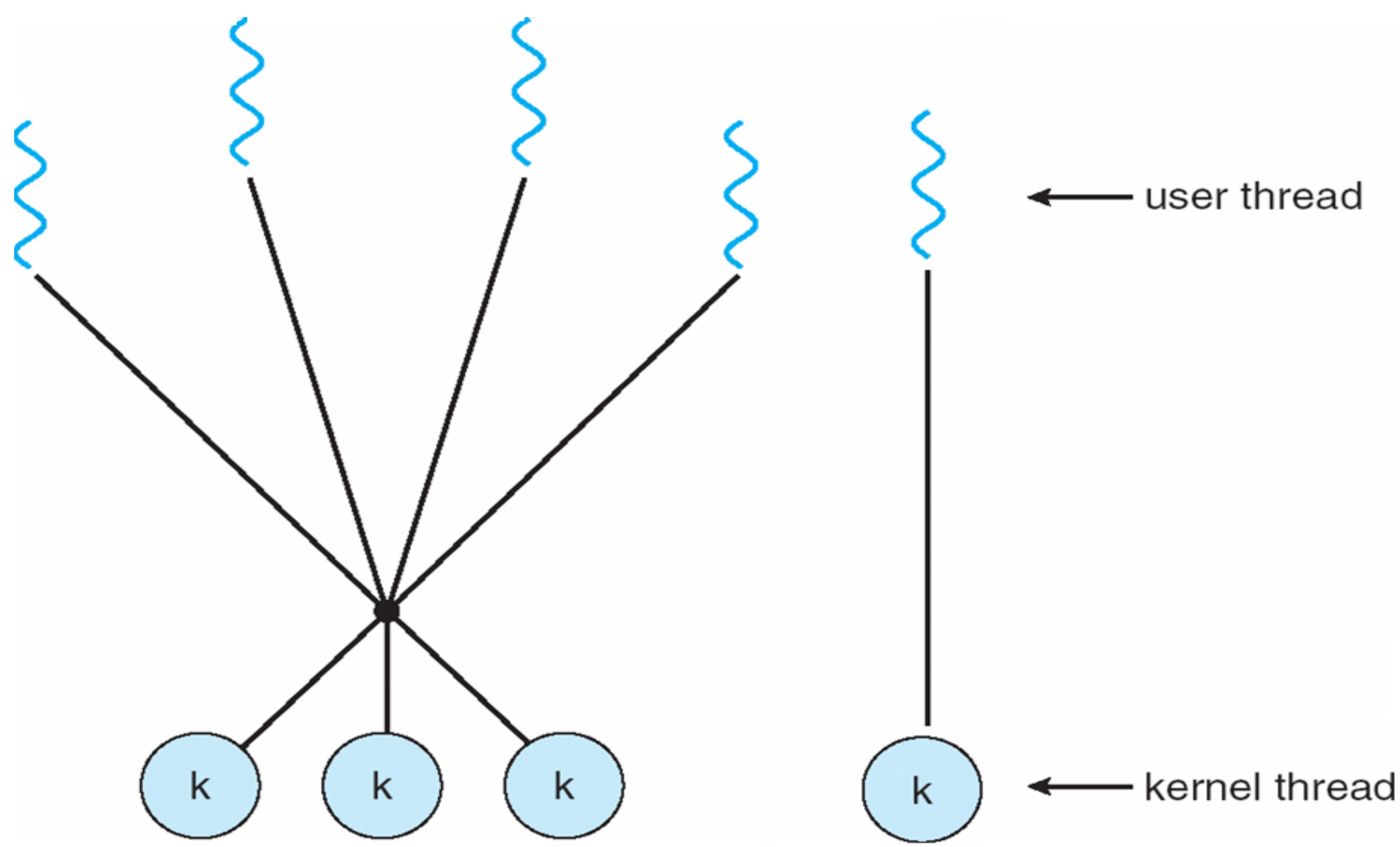
- **Reduces the overhead associated with too many kernel threads as was present in the one-to-one model**



Two-level Model

- **Similar to the Many-to-Many model except that it allows a user thread to be bound to a specific kernel thread**

- **Examples include**
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Thread Libraries

- **A thread library provides programmer with API for creating and managing threads**

- **Two primary ways of implementing**
  - Library entirely in user space
  - Kernel-level library supported by the OS

- **Three main user thread libraries currently in use**

  (1) POSIX PThreads - user-level or kernel-level threads for POSIX-compliant systems

  (2) Win32 threads - kernel-level threads for Windows systems

  (3) Java threads

POSIX Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads

- **Java threads are managed by the JVM**

- **Typically implemented using the threads model provided by underlying OS**

- **Java threads may be created in two different ways:**
  - Extending the Thread class
  - Implementing the Runnable interface

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example (Cont.)

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

Win32 API Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Win32 API Multithreaded C Program (Cont.)

```
    // create the thread
    ThreadHandle = CreateThread(
      NULL, // default security attributes
      0, // default stack size
      Summation, // thread function
      &Param, // parameter to thread function
      0, // default creation flags
      &ThreadId); // returns the thread identifier

    if (ThreadHandle != NULL) {
      // now wait for the thread to finish
      WaitForSingleObject(ThreadHandle,INFINITE);

      // close the thread handle
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
    }
}
```

Figure 4.10 Multithreaded C program using the Win32 API.

Java Multithreaded Program

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}
```

Java Multithreaded Program (Cont.)

```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0)
       System.err.println(args[0] + " must be >= 0.");
      else {
       // create the object to be shared
       Sum sumObject = new Sum();
       int upper = Integer.parseInt(args[0]);
       Thread thrd = new Thread(new Summation(upper, sumObject));
       thrd.start();
       try {
          thrd.join();
          System.out.println
                  ("The sum of "+upper+" is "+sumObject.getSum());
      } catch (InterruptedException ie) { }
      }
     }
     else
      System.err.println("Usage: Summation <integer value>"); }
}
```

Figure 4.11 Java program for the summation of a non-negative integer.